EXHIBIT C

UNITED STATES DISTRICT COURT
NORTHERN DISTRICT OF CALIFORNIA
SAN FRANCISCO DIVISION

ORACLE AMERICA, INC.

Plaintiff,

v.

GOOGLE INC.

Defendant.

Case No. 3:10-cv-03561-WHA

EXPERT REPORT OF DAVID I. AUGUST, PH.D. REGARDING THE NON-INFRINGEMENT OF U.S. PATENT NO. 6,910,205

In other words, Dr. Mitchell does not and cannot explain how the cited materials demonstrate that the Android JIT reads on Asserted Claims 1, 2, 3 and 8.

b. Second, in addition to the above failure, it should also be evident that the Android JIT works in a manner very different than the technique described in the '205 patent and Asserted Claims. Some key distinctions with respect to the Android JIT are:

i. No Dalvik VM instruction is ever new or generated at runtime;

ii. No existing Dalvik VM instruction is ever altered or replaced;

iii. Dalvik VM instructions do not represent or reference native machine instructions – instead, the Dalvik VM itself keeps track of any native machine instructions; and

iv. No new Dalvik VM instruction is ever executed instead of some original Dalvik VM instruction.

F. Claim 1 of the '205 patent (Inline/dexopt Allegations)[4]

199. Having addressed the Asserted Claims with respect to the JIT compiler functionality, I now address the same Asserted Claims but with respect to the dexopt functionality, which the Mitchell Patent Report generally refers to as the inline implementation.

200. Again, claim 1 of the '205 patent reads:

> [Preamble] *In a computer system, a method for increasing the execution speed of virtual machine instructions at runtime, the method comprising:*
>
> [1-a] *receiving a first virtual machine instruction;*
>
> [1-b] *generating, at runtime, a new virtual machine instruction that represents or references one or more native instructions that can be executed instead of said first virtual machine instruction; and*
>
> [1-c] *executing said new virtual machine instruction instead of said first virtual machine instruction.*

[4] Citations to code in this section are from verision 2.2 (Froyo) as of June 27, 2010.

201. Google does not infringe claim 1 of the ‘205 patent because limitations of this claim are not found in Android.

202. With respect to claim limitation 1-b, dexopt does ***not*** process virtual machine instructions at runtime as required in the claim, but rather at installation time. *See* dexopt.html (“[t]he system installer does it when an application is first added.”)

203. In ¶ 400, Dr. Mitchell states “Android uses the Dalvik virtual machine to execute virtual machine bytecode instructions at runtime. The Dalvik virtual machine – and here dexopt in particular – performs and runs code resulting from certain optimizations to increase the execution speed of virtual machine instructions at runtime.” This is misleading.

204. The Dalvik virtual machine executes virtual machine instructions. However, the dexopt routine is not performed by a process that runs applications since that would interfere with its execution by allocating resources that are difficult to release. (*See* dexopt.html.) Instead, dexopt prepares and installs a DEX file for execution before the Dalvik virtual machine is able to run the installed code. In particular, dexopt starts up, completes preparation of the DEX file, installs the ODEX file, and exits – all prior to execution. (*See* dexopt.html.)

/docs/dexopt.html

Preparation

There are at least three different ways to create a "prepared" DEX file, sometimes known as "ODEX" (for Optimized DEX):

There are at least three different ways to create a "prepared" DEX file, sometimes known as "ODEX" (for Optimized DEX):

1. The VM does it "just in time". The output goes into a special `dalvik-cache` directory. This works on the desktop and engineering-only device builds where the permissions on the `dalvik-cache` directory are not restricted. ***On production devices, this is not allowed***.
2. The system installer does it when an application is first added. It has the privileges required to write to `dalvik-cache`.
3. The build system does it ahead of time. The relevant `jar` / `apk` files are present, but the `classes.dex` is stripped out. The optimized DEX is stored next to the original zip archive, not in `dalvik-cache`, and is part of the system image.

The `dalvik-cache` directory is more accurately `$ANDROID_DATA/data/dalvik-cache`.

The files inside it have names derived from the full path of the source DEX. On the device the directory is owned by `system` / `system` and has 0771 permissions, and the optimized DEX files stored there are owned by `system` and the application's group, with 0644 permissions. DRM-locked applications will use 640 permissions to prevent other user applications from examining them. The bottom line is that you can read your own DEX file and those of most other applications, but you cannot create, modify, or remove them.

….

dexopt

We want to verify and optimize all of the classes in the DEX file. The easiest and safest way to do this is to load all of the classes into the VM and run through them. Anything that fails to load is simply not verified or optimized. Unfortunately, this can cause allocation of some resources that are difficult to release (e.g. loading of native shared libraries), so ***we don't want to do it in the same virtual machine that we're running applications in***.

The solution is to invoke a program called `dexopt`, which is really just a back door into the VM. ***It performs an abbreviated VM initialization, loads zero or more DEX files from the bootstrap class path, and then sets about verifying and optimizing whatever it can from the target DEX. On completion, the process exits, freeing all resources***.

It is possible for multiple VMs to want the same DEX file at the same time. File locking is used to ensure that dexopt is only run once.

205. The document cited above notes that dexopt may be used at "just-in-time," but only "on the desktop and engineering-only device builds where the permissions on the dalvik-cache directory are not restricted. ***On production devices, this is not allowed***." (*Id*.)

206. The Dexopt.html document reinforces this concept, noting that "[t]he goals [of the Dalvik virtual machine] led us to make some fundamental decisions: … Optimizations that require rewriting bytecode ***must be done ahead of time***." (*Id*. (emphasis added).)

207. The Mitchell Patent Report quotes from this same document in seeking to demonstrate that dexopt works at "run-time," and suspiciously fails to acknowledge or cite these portions of the document.

208. Oracle's contentions that run-time is "during execution of the virtual machine" also conflicts with Dr. Mitchell's report. (*See* ¶ 102.) Under this construction and the ordinary meaning, dexopt is not performed at runtime. As described above, dexopt exits <u>before</u> the

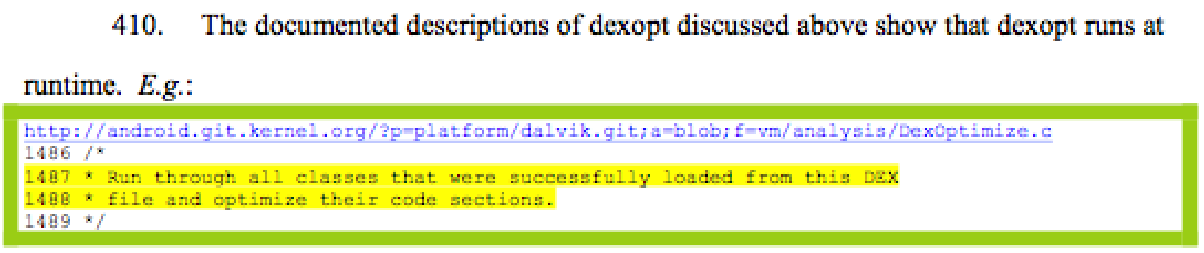
Dalvik virtual machine actually executes the processed virtual machine instructions. (*See* dexopt.html.)

209. The dexopt process compiles an entire method before it is executed. *See* Mitchell Patent Report at ¶ 248 concerning allegations in regard to the '104 patent ("dexopt OptimizeMethod looks at every instruction in a method…"); *see also id*. at ¶ 249:

> The method optimizeMethod excerpted below "[o]ptimize[s] instructions in a method" and "does a single pass through the code, examining each instruction" as the Android developer comments indicate."

See also id. at ¶ 407: "The function optimizeMethod optimizes **instructions** in a method, processing one virtual machine instruction at a time…" (emphasis is his). *See also* lines 1560 to 1680 in /vm/analysis/DexOptimize.c.

210. In ¶ 410, Dr. Mitchell states:



410. The documented descriptions of dexopt discussed above show that dexopt runs at runtime. *E.g.*:

```
http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=vm/analysis/DexOptimize.c
/*
 * Run through all classes that were successfully loaded from this DEX
 * file and optimize their code sections.
 */
```

211. The positions Dr. Mitchell takes with respect to the '104 and '205 patents are inconsistent.

212. If he is referring to the word "run", he confuses the sequential processing of classes dexopt compiles with the runtime of a program. The function referred to by this documented description (reproduced below), shows that the description refers to sequentially processing classes in the DEX file. Each class found is optimized by calling optimizeClass at line 1522. The optimizeClass function does not execute the program in a DEX file.

```
/vm/analysis/DexOptimize.c
/*
 * Run through all classes that were successfully loaded from this DEX
 * file and optimize their code sections.
 */
static void optimizeLoadedClasses(DexFile* pDexFile)
{
    u4 count = pDexFile->pHeader->classDefsSize;
    u4 idx;
    InlineSub* inlineSubs = NULL;

    LOGV("DexOpt: +++ optimizing up to %d classes\n", count);
    assert(gDvm.dexOptMode != OPTIMIZE_MODE_NONE);

    inlineSubs = createInlineSubsTable();

    for (idx = 0; idx < count; idx++) {
        const DexClassDef* pClassDef;
        const char* classDescriptor;
        ClassObject* clazz;

        pClassDef = dexGetClassDef(pDexFile, idx);
        classDescriptor = dexStringByTypeIdx(pDexFile, pClassDef-
>classIdx);

        /* all classes are loaded into the bootstrap class loader */
        clazz = dvmLookupClass(classDescriptor, NULL, false);
        if (clazz != NULL) {
            if ((pClassDef->accessFlags & CLASS_ISPREVERIFIED) == 0 &&
                gDvm.dexOptMode == OPTIMIZE_MODE_VERIFIED)
            {
                LOGV("DexOpt: not optimizing '%s': not verified\n",
                    classDescriptor);
            } else if (clazz->pDvmDex->pDexFile != pDexFile) {
                /* shouldn't be here -- verifier should have caught */
                LOGD("DexOpt: not optimizing '%s': multiple definitions\n",
                    classDescriptor);
            } else {
                optimizeClass(clazz, inlineSubs);

                /* set the flag whether or not we actually did anything */
                ((DexClassDef*)pClassDef)->accessFlags |=
                    CLASS_ISOPTIMIZED;
            }
        } else {
            LOGV("DexOpt: not optimizing unavailable class '%s'\n",
                classDescriptor);
        }
    }

    free(inlineSubs);
}
```

213. Dexopt cannot be running the code because the dexopt routine is performed on bytecode instructions as they are stored in memory order (*i.e.*, as they are stored in memory). By contrast, during actual run-time, the Dalvik VM executes bytecode instructions as they are visited in program order (*i.e.*, the order in which the instructions are actually performed).

214. The accused dexopt process does not practice asserted claim 1 because dexopt is not running during the execution of the virtual machine.

215. As explained above, Claim 1 describes that a "new" virtual machine instruction be "generated" by the method "in a computer system." All virtual machine instructions used by dexopt have a defined meaning and behavior specified beforehand by the developers of the Android system.

216. Claim 1 specifically requires that the virtual machine instruction be "new," which reinforces the concept that the virtual machine instruction does not have any defined meaning or behavior before the application of the method; it is new in the sense that it is just then – as a result of the application of the method – first defined by the generated code snippet in the cache.

217. There is no bytecode used in dexopt that is unassigned or without a defined meaning before the execution of dexopt. Every Dalvik bytecode either has a defined meaning and behavior before runtime or remains unused and illegal. (See ¶ 156.)

218. There are some replacements of virtual machine instructions in dexopt akin to the prior art use quick instructions. But the "generated" and "new" limitations of Claim 1 (on which Claim 2 depends) are separate and distinct from Claim 2's requirement of overwriting an existing virtual machine instruction, and as such are not simply describing a replacement of virtual machine instructions.

219. In ¶ 413, Dr. Mitchel points to OP_EXECUTE_INLINE_RANGE as the new opcode, but he fails to demonstrate that it is *generated* by dexopt. It is, in fact, not new, but a previously existing bytecode with a predefined meaning and behavior.

220. The OP_EXECUTE_INLINE_RANGE *opcode* has a predefined meaning and behavior. OP_EXECUTE_INLINE_RANGE executes a native method. (*e.g.*, /vm/mterp/out/InterpAsm-armv4t.S, lines 7688-7714)

221. The *methods* called by OP_EXECUTE_INLINE_RANGE each have a predefined meaning and behavior. These methods are fixed at development time and small in number. They are String.charAt, String.compareTo, String.equals, String.indexOf (2 variants), String.length, Math.abs (4 variants), Math.min, Math.max, Math.sqrt, Math.cos, Math.sin. These are only methods available to dexopt for use with OP_EXECUTE_INLINE_RANGE. These

methods are listed in the gDVmInlineOpsTable (/vm/InlineNative.c, lines 628-678). These methods are fixed for a given build of the Dalvik VM. In fact, lines 635-636 of vm/InlineNative.c indicate that the Dalvik VM should receive a new DALVIK_VM_BUILD number if this set of methods is changed by developers of the Dalvik VM.

222. The native code versions of the methods listed in gDVmInlineOpsTable (/vm/InlineNative.c, lines 628-678) exist prior to the execution of dexopt (in fact, they exist at the time the Dalvik VM is built).

223. Virtual machine instructions inserted by dexopt have a meaning and behavior defined before dexopt runs. These virtual machine instructions used by dexopt contain the OP_EXECUTE_INLINE_RANGE opcode, an opcode with a predefined meaning and behavior, to call the gDVmInlineOpsTable list of methods, a predefined and fixed set of methods each with a predefined meaning and behavior.

224. OP_EXECUTE_INLINE_RANGE is a quickened instruction. As described above, quickened instructions are acknowledged by ‘205 to be prior art. (*See* ‘205 patent at 2:14-26.) OP_EXECUTE_INLINE_RANGE is used to call a method instead of the ordinary (non-quickened) instructions OP_INVOKE_DIRECT_RANGE, OP_INVOKE_STATIC_RANGE, or OP_INVOKE_VIRTUAL_RANGE. (*See* /vm/analysis/DexOptimize.c comment at 2347 and assertion at 2366-2368)

vm/analysis/DexOptimize.c

```
   2345 /*
   2346  * See if the method being called can be rewritten as an inline operation.
   2347  * Works for invoke-virtual/range, invoke-direct/range, and invoke-
static/range.
   2348  *
   2349  * Returns "true" if we replace it.
   2350  */
   2351 static bool rewriteExecuteInlineRange(Method* method, u2* insns,
   2352     MethodType methodType, const InlineSub* inlineSubs)
   2353 {
   2354     ClassObject* clazz = method->clazz;
   2355     Method* calledMethod;
   2356     u2 methodIdx = insns[1];
   2357
   2358     calledMethod = dvmOptResolveMethod(clazz, methodIdx, methodType, NULL);
   2359     if (calledMethod == NULL) {
   2360         LOGV("+++ DexOpt inline/range: can't find %d\n", methodIdx);
   2361         return false;
   2362     }
   2363
   2364     while (inlineSubs->method != NULL) {
   2365         if (inlineSubs->method == calledMethod) {
   2366             assert((insns[0] & 0xff) == OP_INVOKE_DIRECT_RANGE ||
   2367                    (insns[0] & 0xff) == OP_INVOKE_STATIC_RANGE ||
   2368                    (insns[0] & 0xff) == OP_INVOKE_VIRTUAL_RANGE);
   2369             insns[0] = (insns[0] & 0xff00) | (u2) OP_EXECUTE_INLINE_RANGE;
   2370             insns[1] = (u2) inlineSubs->inlineIdx;
   2371 
   2372             //LOGI("DexOpt: execute-inline/range %s.%s --> %s.%s\n",
   2373             //    method->clazz->descriptor, method->name,
   2374             //    calledMethod->clazz->descriptor, calledMethod->name);
   2375             return true;
   2376         }
   2377 
   2378         inlineSubs++;
   2379     }
   2380 
   2381     return false;
   2382 }
   2383 
```

225. The authors of the Dalvik VM refer to the OP_EXECUTE_INLINE_RANGE as a quickened instruction. *See* /vm/analysis/CodeVerify.c, lines 5407-5453:

```
5382     /*
5383      * Verifying "quickened" instructions is tricky, because we have
5384      * discarded the original field/method information.  The byte offsets
5385      * and vtable indices only have meaning in the context of an object
5386      * instance.
5387      *
5388      * If a piece of code declares a local reference variable, assigns
5389      * null to it, and then issues a virtual method call on it, we
5390      * cannot evaluate the method call during verification.  This situation
5391      * isn't hard to handle, since we know the call will always result in an
5392      * NPE, and the arguments and return value don't matter.  Any code that
5393      * depends on the result of the method call is inaccessible, so the
5394      * fact that we can't fully verify anything that comes after the bad
5395      * call is not a problem.
5396      *
5397      * We must also consider the case of multiple code paths, only some of
5398      * which involve a null reference.  We can completely verify the method
5399      * if we sidestep the results of executing with a null reference.
5400      * For example, if on the first pass through the code we try to do a
5401      * virtual method invocation through a null ref, we have to skip the
5402      * method checks and have the method return a "wildcard" type (which
5403      * merges with anything to become that other thing).  The move-result
5404      * will tell us if it's a reference, single-word numeric, or double-word
5405      * value.  We continue to perform the verification, and at the end of
5406      * the function any invocations that were never fully exercised are
5407      * marked as null-only.
5408      *
5409      * We would do something similar for the field accesses.  The field's
5410      * type, once known, can be used to recover the width of short integers.
5411      * If the object reference was null, the field-get returns the "wildcard"
5412      * type, which is acceptable for any operation.
5413      */
5414     case OP_EXECUTE_INLINE:
5415     case OP_EXECUTE_INLINE_RANGE:
5416     case OP_INVOKE_DIRECT_EMPTY:
5417     case OP_IGET_QUICK:
5418     case OP_IGET_WIDE_QUICK:
5419     case OP_IGET_OBJECT_QUICK:
5420     case OP_IPUT_QUICK:
5421     case OP_IPUT_WIDE_QUICK:
5422     case OP_IPUT_OBJECT_QUICK:
5423     case OP_INVOKE_VIRTUAL_QUICK:
5424     case OP_INVOKE_VIRTUAL_QUICK_RANGE:
5425     case OP_INVOKE_SUPER_QUICK:
5426     case OP_INVOKE_SUPER_QUICK_RANGE:
5427         failure = VERIFY_ERROR_GENERIC;
5428         break;
```

See also /vm/analysis/RegisterMap.c, lines 2992-3016:

```
/vm/analysis/RegisterMap.c
   2992     /*
   2993      * See comments in analysis/CodeVerify.c re: why some of these are
   2994      * annoying to deal with.  It's worse in this implementation, because
   2995      * we're not keeping any information about the classes held in each
   2996      * reference register.
   2997      *
   2998      * Handling most of these would require retaining the field/method
   2999      * reference info that we discarded when the instructions were
   3000      * quickened.  This is feasible but not currently supported.
   3001      */
   3002     case OP_EXECUTE_INLINE:
   3003     case OP_EXECUTE_INLINE_RANGE:
   3004     case OP_INVOKE_DIRECT_EMPTY:
   3005     case OP_IGET_QUICK:
   3006     case OP_IGET_WIDE_QUICK:
   3007     case OP_IGET_OBJECT_QUICK:
   3008     case OP_IPUT_QUICK:
   3009     case OP_IPUT_WIDE_QUICK:
   3010     case OP_IPUT_OBJECT_QUICK:
   3011     case OP_INVOKE_VIRTUAL_QUICK:
   3012     case OP_INVOKE_VIRTUAL_QUICK_RANGE:
   3013     case OP_INVOKE_SUPER_QUICK:
   3014     case OP_INVOKE_SUPER_QUICK_RANGE:
   3015         dvmAbort();     // not implemented, shouldn't be here
   3016         break;
```

226. The accused dexopt process does not practice asserted claim 1 because it does not generate new virtual machine instructions.

227. As explained above, Claim 1 specifically requires that the virtual machine instruction be "generated," which means that the virtual machine instructions does not have a defined meaning or behavior before runtime.

228. Claim 1 specifically requires that the virtual machine instruction be "new," which reinforces the concept that the virtual machine instruction does not have any defined meaning or behavior before runtime; it is new in the sense that it is just then – at runtime – first defined by the code snippet in the cache.

229. There is no bytecode used in dexopt that is unassigned or without a defined meaning before runtime. Every Dalvik bytecode either has a defined meaning and behavior before runtime or remains unused and illegal.

230. There are some replacements of virtual machine instructions in dexopt akin to the prior art use quick instructions. But the "generated" and "new" limitations of Claim 1 (on which Claim 2 depends) are separate and distinct from Claim 2's requirement of overwriting an existing virtual machine instruction, and as such are not simply describing a replacement of virtual machine instructions.

231. To be clear, OP_EXECUTE_INLINE_RANGE is just like any other bytecode instruction – its behavior is pre-defined. It also is executed like other instructions – with fixed native code instructions that are not encoded in a cache. (They may be in a static table, but so are the native machine code *for all of the other virtual machine instructions*.) Unlike the '205 patent, dexopt cannot replace arbitrary sequences of code, it can only "quicken" the invocation of the 15 functions identified in ¶ 415 of the Mitchell Patent Report. If those 15 functions aren't used, dexopt cannot use OP_EXECUTE_INLINE_RANGE.

232. The Mitchell Patent Report acknowledges this when noting in ¶ 420 that the new bytecode does not directly point to any native instructions. The bytecode simply points to an index table like all other bytecode instructions. Separately, a lookup (dispatch) routine must be performed on the index table to identify relevant native code, which can then be executed.

233. In view of the foregoing, the accused dexopt process does not practice asserted Claim 1 because it does not have a "new virtual machine instruction that represents or references one or more native instructions."

234. As noted above, dexopt generally runs at installation time, not run-time. Even when it is run *after* installation time, the DEX file bytecode is only optimized (*i.e.*, modified) during the dexopt process, but it is ***not*** executed by dexopt.

235. In other words, the dexopt routine is not performed by the Dalvik virtual machine running applications since that would interfere with its execution by allocating resources that are difficult to release. (*See* dexopt.html.) Instead, dexopt prepares a DEX file for execution *before* the Dalvik virtual machine is able run the installed code. In particular, dexopt starts, completes

the preparation of the DEX file, writes the ODEX, and exits all prior to execution of the instructions being processed. (*See* dexopt.html.)

236. The accused dexopt process does not practice asserted claim 1 because there is no "executing said new virtual machine instruction" as required in claim 1-c. The dex file bytecode is optimized (*i.e.*, modified) during the dexopt process, but it is ***not*** executed.

G. Claim 2 of the '205 patent (Inline/dexopt Allegations)

237. Claim 2 of the '205 patent reads:

> *The method of claim 1, further comprising overwriting a selected virtual machine instruction with a new virtual machine instruction,*
>
> *the new virtual machine instruction specifying execution of the at least one native machine instruction.*

238. Google has not infringed claim 2 of the '205 patent because limitations of this claim are not found in Android.

239. As explained above in ¶ **Error! Reference source not found.** –235, new bytecode instructions in dexopt do not point to native code (they reference a look up table), nor do they actually execute any native instructions, because none of the program bytecode is executed in dexopt.

240. Further, as Claim 2 is a dependent claim, extending from independent Claim 1, it is not infringed for the reasons stated above with respect to Claim 1, which are incorporated by reference herein.

H. Claim 3 of the '205 patent (Inline/dexopt Allegations)

241. Claim 3 of the '205 patent reads:

> *The method of claim 2, wherein the new virtual instruction includes a pointer to the at least one native machine instruction.*

242. Google has not infringed claim 3 of the '205 patent because limitations of this claim are not found in Android.

243. As explained above in ¶ **Error! Reference source not found.** –235, new bytecode instructions in dexopt do ***not*** point to native code (they reference a look up table).

244. The accused dexopt process does not practice asserted claim 3 because "the new virtual machine instruction" does ***not*** "include[] ... at least one native machine instruction."

245. Further, as Claim 3 is a dependent claim, extending from dependent Claim 2, which extends from independent Claim 1, it is not infringed for the reasons stated above with respect to Claims 1 and 2, which are incorporated by reference herein.

I. Claim 8 of the '205 patent (Inline/dexopt Allegations)

246. Claim 8 of the '205 patent reads:

> [Preamble] *In a computer system, a method for increasing the execution speed of virtual machine instructions, the method comprising:*
>
> [8-a] *inputting virtual machine instructions for a function;*
>
> [8-b] *compiling a portion of the function into at least one native machine instruction so that the function includes both virtual and native machine instruction;*
>
> [8-c] *representing said at least one native machine instruction with a new virtual machine instruction that is executed after the compiling of the function.*

247. Google has not infringed Claim 8 of the '205 patent because limitations of this claim are not found in Android.

248. Dr. Mitchell takes no position as to whether Claim 8 is literally infringed.

249. Dr. Mitchell takes no position as to whether Claim 8 is infringed under the doctrine of equivalence.

J. Other Observations

250. There are a few concepts that the Mitchell Patent Report attempts to equate with the '205 patent:

a. It claims that "[a]n important idea in the'205 patent is that some of the bytecodes associated with a function may be compiled." *Id*. at ¶ 25. However, as discussed above, the concept of hybrid systems and just-in-time compiling of bytecode instructions has long been known and described in the prior art. *See, e.g.*, Exhibits K–Q.

Executed this 25th day of August, 2011.

I declare that to the best of my knowledge the foregoing is true and correct as to the facts stated and my opinions as expressed.

[illegible]

David I. August, Ph.D.